

A Guide to Commenting

Version 3 – March, 2008

A few small changes

Version 2 – Feb, 2006

Many small changes

Version 1 – Jun, 2001

Initial release

Jack G. Ganssle
jack@ganssle.com

The Ganssle Group
PO Box 38346
Baltimore, MD 21231
(410) 504-6660
fax (647) 439-1454

According to Henry Petroski (reference 1), the first known book about engineering is the 2000 year old work “De Architectura” by Marcus Vitruvius Pollio. It’s a fairly complete description of how these skilled artisans created their bridges and tunnels in ancient Rome.

One historian said of Vitruvius and his book: “He writes in atrocious Latin, but he knows his business”. Another wrote: “He has all the marks of one unused to composition, to whom writing is a painful task”.

How little things have changed! Even two millennia ago engineers wrote badly, yet were recognized as experts in their field. Perhaps even then these Romans were geeks. Were engineers from Athens Greek geeks?

Some developers care little about their poor writing skills, figuring they interact with machines, not people. And of course we developers just talk to other writing-challenged engineers anyway, right?

Wrong.

This is the communications age. The spoken and written word has never been more important. Consider how email has reinvigorated letter-writing.... yet years ago I remember hearing philologists moaning about the death of letters.

Old timers will remember how engineers could once function perfectly with no typing skills. That seems quaint today, when most of us live with a keyboard all but strapped to our hands. Just as old-fashioned is the idea of a secretary transcribing notes and the fixing spelling and grammar. Today it’s up to *us* to express ourselves clearly, with only the assistance of a spellchecker and an annoyingly-picky grammar engine.

I write a weekly column on embedded.com which generates quite a bit of feedback via email. The majority of these responses are quite well written, giving lie to the old generalization of engineers being compositionally challenged. But some replies are rather appalling. Obviously non-English speakers struggle with our language’s idiosyncrasies. But all too many of these confusing ungrammatical missives come from Joe Smith in Anytown, USA.

Even if you’re stuck in a hermitically-sealed cubicle never interacting with people and just cranking code all day, I contend you still have a responsibility to communicate clearly and grammatically with others. Software is, after all, a mix of computerese (the C or C++ itself) and comments (in America, at least, an English-language description meant for humans, not the computer). If we write perfect C with illegible comments, we’re doing a lousy job.

I read a *lot* of code from a huge range of developers. Consistently well-done comments are rare. Sometimes I can see the enthusiasm of the team at the project’s outset. The

© 2006 The Ganssle Group. This work may be used by individuals and companies, but all publication rights reserved.

startup code is fantastic. Main()'s flow is clear and well documented. As the project wears on functions get added and coded with less and less care. Comments like

```
/* ???? */
```

or my favorite:

```
/* Is this right? */
```

start to show up. Commenting frequency declines; clarity gives way to short cryptic notes; capitalization descends into chaotic randomness. The initial project excitement, as shown in the careful crafting of early descriptive comments, yields to schedule panic as the developers all but abandon anything that's not executable.

Onerous and capricious schedules are a fact of life in this business. It's natural to chuck everything not immediately needed to make the product work. Few bosses grade on quality of the source code. Quality, when considered at all, is usually a back-end complaint about all the bugs that keep surfacing in the released product, or the ongoing discovery of defects that pushes the schedule back further and further.

Pride

We firmware folks know that quality starts at the front-end, in proper design and implementation, using reasonable processes. Quality also requires fine workmanship. Our profession parallels that of the trade crafts of centuries ago. The perfect joint in a chair may be almost invisible, but will last forever. A shoddy alternative could be just as hard to see, but is simply not acceptable. Professional pride mandates doing the right thing just because we know it's the best way to build the product.

Most of us create software in secret. I rarely see companies using code inspections, for example, which at the very least brings our flaws into the cold harsh light of day. Secrecy naturally breeds laziness. It takes a very strong person to consistently rise above the temptations of expediency to do things *right*, even when it's not clear that there will be a reward for working carefully.

Though we embedded people work at the border between hardware and software, where sometimes it's hard to say where one ends and the other starts, even hardware designers work in the spotlight. Their creations are subject to ongoing audits during manufacturing, test and repair. Technicians work with the schematics daily. Faults glare from the page for everyone to see. Sloppy work can't be hidden.

(Now, though, ASICs, programmable logic and high level synthesis can bury lots of evil in the confines of an inscrutable IC package. The hardware folks are inheriting all of the perils of software.)

I'm fascinated by eXtreme Programming, though shudder at some of the practices it espouses. All of XP's ideas come from four "core values": communications, simplicity, feedback and courage. No other methodology that I'm aware of derives from *values*. In America we talk a lot about values, sometimes so much so that the meaning gets lost in the rhetoric. Yet values of all sorts are the basis of good behavior. I think the XP folks got it right by deriving the process from values rather than from a collection of good ideas. However, I'd add a fifth to their list: Pride of Workmanship.

In my experience software created without pride is awful. Shortcuts abound. The limited docs never mirror current reality. Error conditions and exceptions are poorly thought-out. For example, Microsoft's various products have garnered a reputation for their susceptibility to buffer overflow attacks. Unix, too, has long suffered the same flaws. Recent posts on the Risks forum <http://catless.ncl.ac.uk/Risks/21.84.html> and <http://catless.ncl.ac.uk/Risks/21.85.html> suggest that the C language is the source of the problem. Programs written in C usually have no intrinsic array bounds checking; worse, the dynamic nature of pointers makes automatic run time checks that much more problematic.

I disagree. C is nothing more than a tool, one that should come with an "adults only" warning. Those who use it carelessly are at fault, not the language itself. Index into a data structure without adding the requisite overflow checks and you're playing with dynamite. While smoking. In a puddle of gasoline.

Every programmer knows he or she should run simple sanity checks on all data from untrusted sources. Not doing so is laziness, a lack of Pride in Workmanship. Careful craftsmen spend a few seconds adding these checks to save months of debugging or millions in product recalls.

Commenting Suggestions

My standard for commenting is that someone versed in the functionality of the product – but not the software – should be able to follow the program flow by reading the comments without reference to the code itself. Code implements an algorithm; the comments communicate the code's operation to yourself and others. Maybe even to a future version of yourself during maintenance years from now.

Write every bit of the documentation (in the USA at least) in English. Noun, verb. Use active voice. Be concise; don't write the Great American Novel. Be explicit and complete; assume your reader hasn't the slightest insight into the solution of the problem. In most cases I prefer to incorporate an algorithm description in a function's header, even for well-known approaches like Newton's Method. A description that uses your variable names makes a lot more sense than "see any calculus book for a description." And let's face it: once carefully described in the comments it's almost trivial to implement the code.

Capitalize per standard English procedures. IT HASN'T MADE SENSE TO WRITE ENTIRELY IN UPPER CASE SINCE THE TELETYPE DISAPPEARED 25 YEARS AGO. the common c practice of never using capital letters is also obsolete. Worst aRe the DevElopeRs wHo uSE rAndOm caSe changeS. Sounds silly, perhaps, but I see a lot of this. And spel al of the wrds.

Avoid long paragraphs. Use simple sentences. "Start_motor actuates the induction relay after a pause of <PAUSETIME> seconds, where <PAUSETIME> is defined in HEADER.H" beats "this function, when called, will start it all off and flip on the external controller but not until a time defined in HEADER.H goes by."

Begin every module and function with a header in a standard format. The format may vary a lot between organizations, but should be consistent within a team. Every module (source file) must start off with a general description of what's in the file, the company name, a copyright message if appropriate, and dates. Start every function with a header that describes what the routine does and how, goes-intas and goes-outas (i.e., parameters), the author's name, date, version, a record of changes with dates and the name of the programmer who made the change.

C lends itself to the use of asterisks to delimit comments, which is fine. I see a lot of this:

```
/*  
 * comment  
*/
```

which is a lousy practice. If your comments end with an asterisk as shown, every edit requires fixing the position of the trailing asterisk. Leave it off, as follows:

```
/*  
 comment  
*/
```

Most modern C compilers accept C++'s double slash comment delimiters, which is more convenient than the /* */ C requires. Start each comment line with the double slash so the difference between comments and code is crystal clear.

Some folks rely on a fancy editor to clean up comment formatting or add trailing asterisks. Don't. Editors are like religion. Everyone has their own preference, each of which is configured differently. Someday compilers will accept source files created with a word processor which will let us define editing styles for different parts of the program. Till then dumb ASCII text formatted with spaces (not tabs) is all we can count on to be portable and reliable.

Enter comments in C at block resolution and when necessary to clarify a line. Don't feel compelled to comment each line. It is much more natural to comment groups of lines which work together to perform a macro function.

Explain the meaning and function of every variable declaration. Long variable names are merely an *aid* to understanding; accompany the descriptive name with a deep, meaningful, prose description. For instance:

```
uint8      Encoder; // Current encoder position; set by
              // the encoder ISR.
```

Exploring the naming issue a bit more, insure that names start with the big and work to the small. An example is: Universe_Galaxy_SolarSystem_Planet. For example:

```
Timer_0_Initialize
```

is better than:

```
Initialize_Timer_0
```

If you were looking through a dictionary or link map that lists variable names, you're more likely to focus on functions related to the timer, rather than to initializing things. So for a timer we might find:

```
Timer_0_Initialize
Timer_0_Read
Timer_0_Set
```

Secondly, never use acronyms and abbreviations as part of a variable or function name, unless such acronym/abbreviation is defined in the code in a special abbreviations table, or if it's an accepted industry convention like LED, LCD, and CRT.

Clarity is our goal! Where "Disp" might mean display (as a verb) to you, to someone else it might imply a chunk of hardware. "Enc" could be encode or encoder.

An example Abbreviation Table is:

```
/* Abbreviation Table
Dsply    == Display (the verb)
Disp     == Display (our LCD display)
Tot      == Total
Calc     == Calculation
Pos      == Position
*/
```

In assembly language feel free to use comments that start in column 1, as well as those appended after an instruction, as follows:

```
; See if the token pointed to by BX is an ASCII hex char
  mov  al,[bx]    ; AL contains the token
  cmp  al,'0'     ; Is it a zero or bigger?
  jns  exit       ; Branch if less than '0'
```

But in C or C++ avoid the use of comments to the right of code... because such practice results in code that looks like hell and is hard to read. The comments never stand out from the C itself. Instead, with the exception of #DEFINES, variable declarations and the like, start all comments on their own line in column 1.

One of the perils of good comments – which is frequently used as an excuse for sloppy work – is that over time the comments no longer reflect the truth of the code. Comment drift is intolerable. Pride in Workmanship means we change the docs as we change the code. The two things happen in parallel. Never defer fixing comments till later, as it just won't happen. Better: edit the descriptions first, then fix the code.

One side effect of our industry's inglorious 50 year history of comment drift is that people no longer trust comments. Such lack of confidence leads to even sloppier work. It's hard to thwart this descent into commenting chaos. Wise developers edit the header to reflect the update for each patch, but even better add a note that says "comments updated, too" to build trust in the docs, as follows:

```
/*
*****

Function int Read_AtoD(void)

Version 1.0 - Initial release 11/4/2005 by Bill Coder

Version 1.1 - Added time-out code in case
end-of-convert never comes.
12/1/2005 by Jill Developer
Updated the comments to reflect the
changes in the code.

*****
*/
```

A code terrorist can block copy the "comments updated to reflect changes in the code" statement... but most of us are a decent sort. We use this as a crutch to help us remember to update the comments and build trust in the comments for future readers of the code.

If you use code inspections (and please do – they are the cheapest known way to get rid of bugs. See <http://ganssle.com/inspections.htm> for a description) review the comments as well as the code. Both are equally important.

A Guide to Commenting

Consider changing the way you write functions. Write *all* of the comments first, including the header and those buried in the code. Then it's simple, even trivial, to fill in the C or C++. Any idiot can write software following a decent design; inventing the design, reflected in well-written comments, is the really creative part of our jobs.

Finally, remember and practice the Golden Rule: “document unto others as you would have documented unto you.”

Reference 1: The Pencil : A History of Design and Circumstance by Henry Petroski (December 1992) Knopf; ISBN: 0679734155